Escuela de Computación

**Análisis de algoritmos**

Tarea Programada #1

Estudiantes: Erick Quesada Fonseca

José Barrientos Rojas

Geovanny Astorga López

Silvia Calderón Navarro

Contenido

[Análisis de algoritmos: 2](#_Toc495871012)

[1. Algoritmo “Vuelto de monedas”: 2](#_Toc495871013)

[2. Algoritmo de la mochila: 2](#_Toc495871014)

[3. Floyd y Dijkstra: 2](#_Toc495871015)

[4. Torres de Hannoi: 2](#_Toc495871016)

[5. Quicksort y Heapsort: 2](#_Toc495871017)

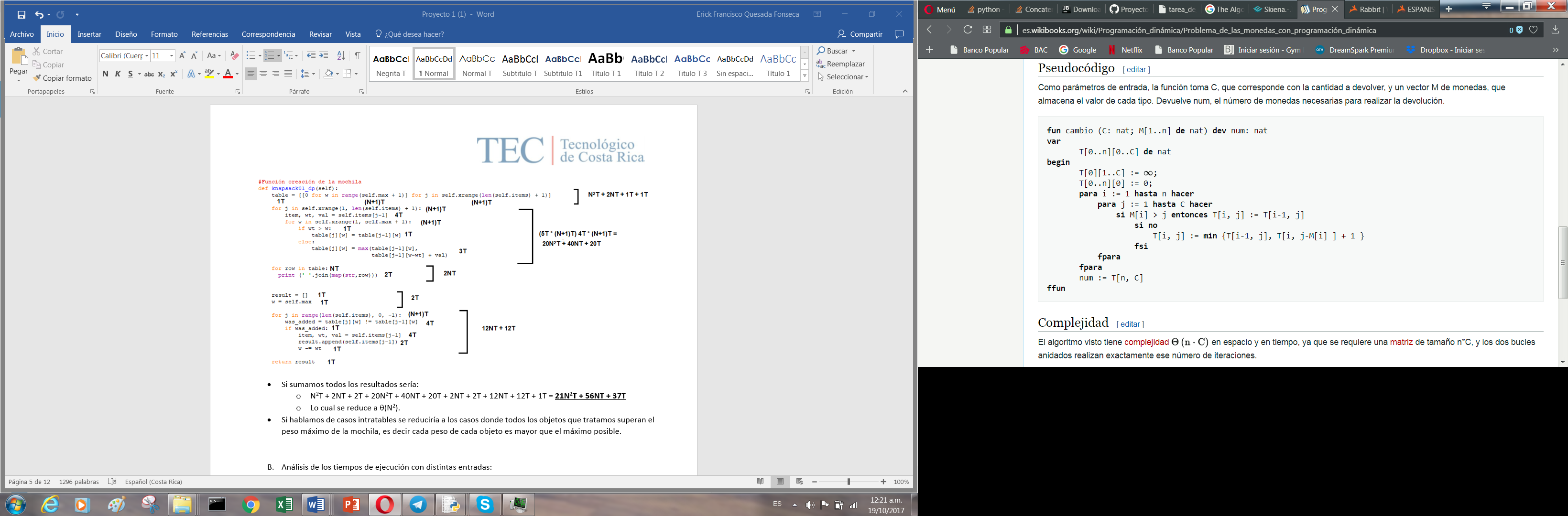
[6. Multiplicación de matrices en cadena: 2](#_Toc495871018)

[Preguntas: 2](#_Toc495871019)

# Análisis de algoritmos:

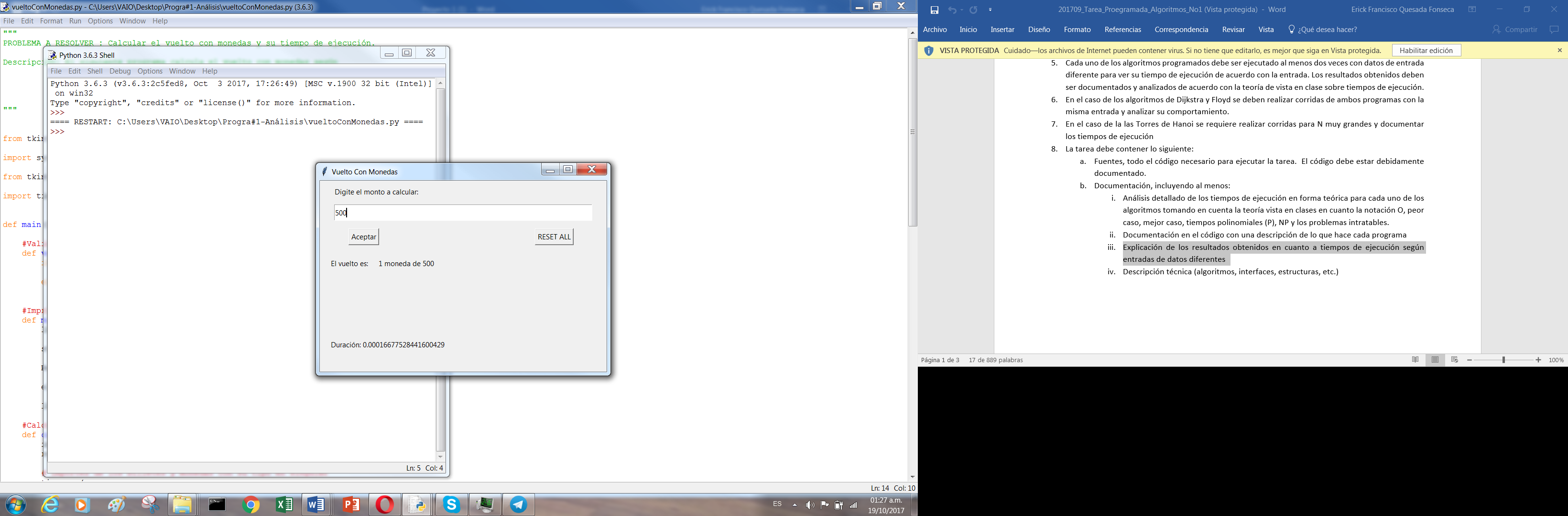
1. Algoritmo “Vuelto de monedas”:

En el algoritmo de Vuelto con Monedas se tienen monedas con diferentes valores y se debe devolver una cantidad correspondiente al valor determinado. Siguiendo el método de la programación dinámica, se rellenará una [tabla](https://es.wikibooks.org/w/index.php?title=Tabla&action=edit&redlink=1) con las filas correspondientes a cada valor para las monedas y las columnas con valores desde el 1 hasta el valor dado. Cada posición (i, j) de la tabla nos indica el número mínimo de monedas requeridas para devolver la cantidad j con monedas con valor menor o igual al de i.

1. Para cada casilla de la tabla hacer:
2. Si el valor de la moneda actual es mayor que la cantidad, se paga con el resto de monedas, es decir, se toma el resultado de la casilla superior.
3. Si el valor de la moneda actual es menor o igual que la cantidad, se toma el mínimo entre:
   1. Pagar con el resto de monedas, tomando el resultado de la casilla superior.
   2. Pagar con una moneda del tipo actual y el resto con el resultado que se hubiera obtenido al pagar la cantidad actual a la que se le ha restado el valor de la moneda actual.
4. Tomar como resultado el valor de la última celda.

El algoritmo visto tiene complejidad **θ(n · C)** en espacio y en tiempo, ya que se requiere una matriz de tamaño n\*C, y los dos bucles anidados realizan exactamente ese número de iteraciones.

La complejidad espacial puede reducirse a **θ(C)** ya que, en cada iteración, sólo necesitamos los datos almacenados en dos filas de la matriz (las filas i e i-1). El mejor caso sería que el monto a devolver sea igual a la primer moneda a consultar. Y el peor caso es cuando para devolver el monto ocupe 1 o más cantidad de cada una de las monedas.

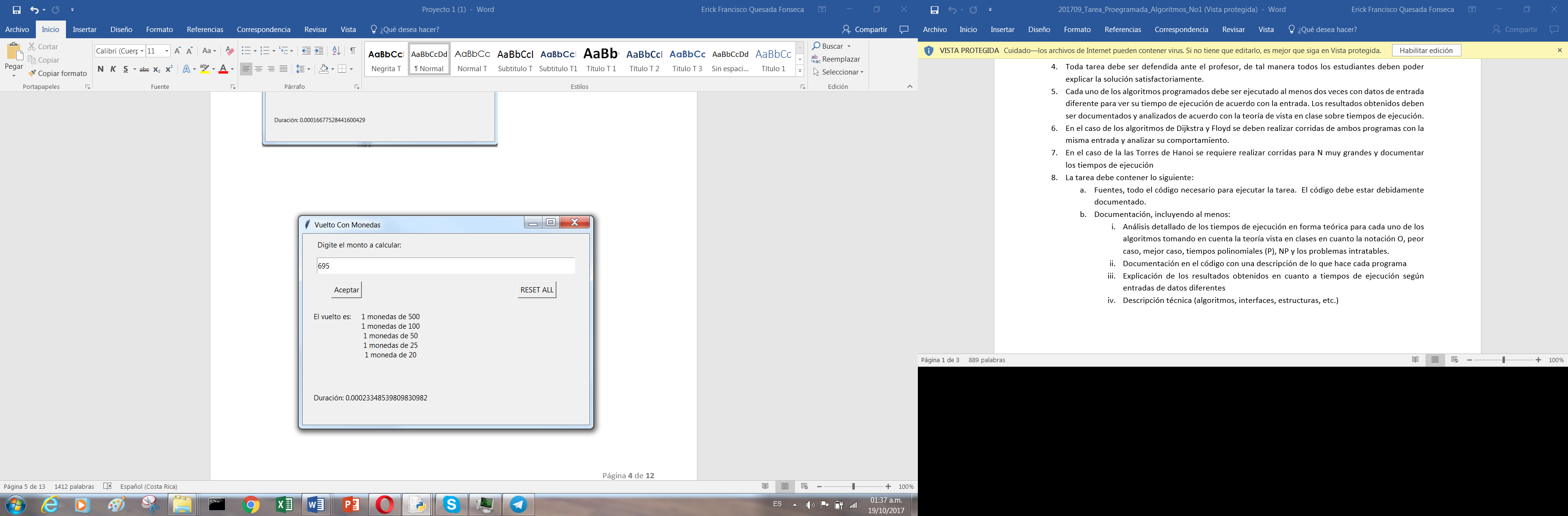


**Monto a calcular:** 500

**Vuelto:** 1 moneda de 500

0 monedas de las demás categorías.

**Tiempo:** 0.0001667s



**Monto a calcular:** 695

**El vuelto es:** 1 de 500

1 de 100

1 de 50

1 de 25

1 de 20

**Tiempo:** 0.0002334s

<https://es.wikibooks.org/wiki/Programación_dinámica/Problema_de_las_monedas_con_programación_dinámica>

1. Algoritmo de la mochila:

El algoritmo de la mochila es un problema que busca la mejor solución entre un conjunto finito de posibles soluciones. En este caso se tiene una mochila con una capacidad de peso máximo, y un conjunto de objetos con un peso y un valor cada uno. La idea del problema de la mochila es encontrar aquellos elementos que maximicen el valor sin exceder el peso máximo. En este caso se tratará con una mochila 0-1, lo cual significa que se tiene un solo elemento de cada tipo.

1. Análisis de los tiempos de ejecución en forma teórica:

* La decisión de incluir o descartar un elemento es del tipo NP completo, ya que no se conoce un algoritmo que sea correcto y rápido para todos los casos.
* El problema de optimización es un NP-duro, su solución es como mínimo tan difícil como la decisión del problema (la decisión de incluir o descartar un elemento), y no se conoce un algoritmo que dada una solución pueda decir si es óptima o no.
* Existe un algoritmo que produce un tiempo pseudo-polinomial utilizando programación dinámica, en este caso ese el algoritmo que se va a utilizar.
* Tiempo de ejecución: θ(NW).

Análisis:



* Si sumamos todos los resultados sería:
  + N2T + 2NT + 2T + 20N2T + 40NT + 20T + 2NT + 2T + 12NT + 12T + 1T = **21N2T + 56NT + 37T**
  + Lo cual se reduce a θ(N2).
* Si hablamos de casos intratables se reduciría a los casos donde todos los objetos que tratamos superan el peso máximo de la mochila, es decir cada peso de cada objeto es mayor que el máximo posible.

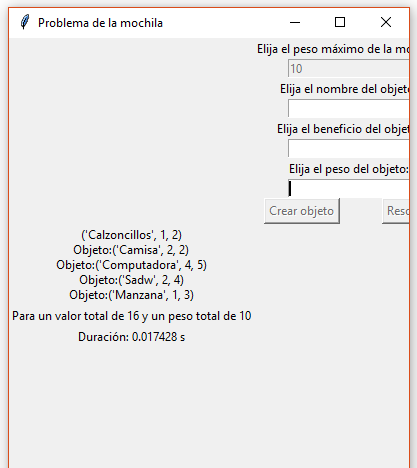
1. Análisis de los tiempos de ejecución con distintas entradas:

|  |  |  |
| --- | --- | --- |
| Nombre | Peso | Beneficio |
| Manzana | 1 | 3 |
| Sandwich | 2 | 4 |
| Computadora | 5 | 4 |
| Camisa | 2 | 2 |
| Pantalón | 3 | 2 |
| Coca Cola | 1 | 1 |
| Calzoncillos | 1 | 2 |

Peso máximo: 10

Elementos: 7

Tiempo: 0.017428s



Máximo peso: 5

Elementos: 7

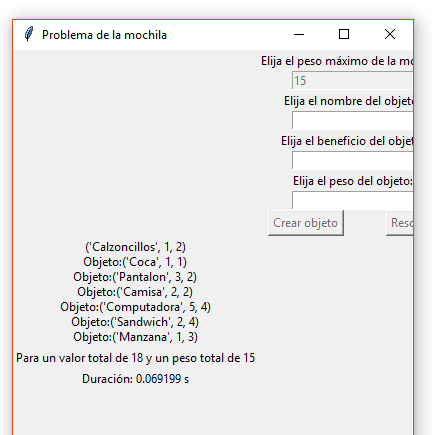
Tiempo: 0.018117s



Máximo peso: 15

Elementos: 7

Tiempo: 0.069199s

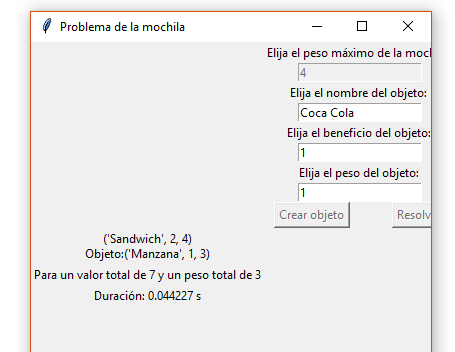


|  |  |  |
| --- | --- | --- |
| Nombre | Peso | Beneficio |
| Manzana | 1 | 3 |
| Sandwich | 2 | 4 |
| Coca Cola | 1 | 1 |

Máximo peso: 4

Elementos: 3

Tiempo: 0.044227s

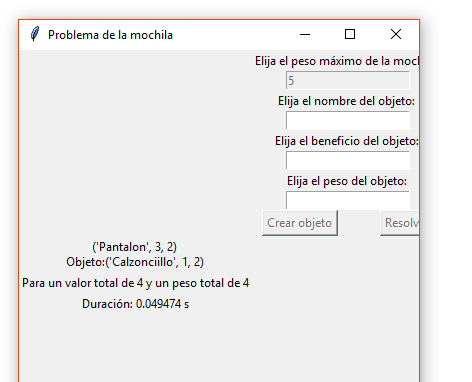


|  |  |  |
| --- | --- | --- |
| Nombre | Peso | Beneficio |
| Calzoncillos | 1 | 2 |
| Pantalón | 3 | 2 |
| Camisa | 2 | 2 |
| Computadora | 5 | 4 |

Máximo peso: 5

Elementos: 4

Tiempo: 0.049474s



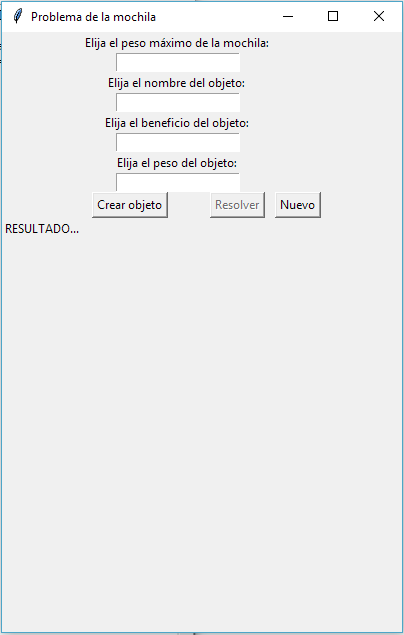
1. Descripción técnica:

**Algoritmo y estructura:**

Se utiliza un algoritmo de programación dinámica no recurrente, el cuál utiliza las siguientes estructuras:

* Un arreglo en el que se guarda la información de cada ítem o elemento, el nombre, el peso y el beneficio.
* Un arreglo bidimensional para guardar los valores máximos.
* Por último, se utiliza un arreglo para guardar los objetos escogidos después de analizar la matriz anterior. Estos elementos serán el resultado final del algoritmo.

**Interfaz:**



**1**

**2**

**Descripción:**

**7**

**6**

**5**

**3**

**4**

1. El primer paso es introducir el peso máximo de la mochila. Esto se realizará una única vez con cada mochila, al crear el primer objeto.
2. Al crear un objeto, se debe elegir un nombre que lo identifique, se recomienda que este no se repita, para poder diferenciar los objetos apropiadamente.
3. Cada objeto debe contar con un beneficio que lo caracteriza, este tiene que ser un valor numérico mayor que cero, en caso contrario el programa no permitirá que se introduzca.
4. Cada objeto debe contar con un peso que lo caracteriza, este tiene que ser un valor numérico mayor que cero, en caso contrario el programa no permitirá que se introduzca.
5. El botón crear objeto, permite añadir un objeto a la lista de elementos disponibles para la mochila.
6. Resuelve el algoritmo de la mochila, devolviendo el tiempo de ejecución en segundos, la lista de objetos utilizados, el peso total que se obtuvo de estos y el beneficio total.
7. El botón nuevo limpia la lista de objetos y crea una nueva mochila.
8. Dijkstra y Floyd:

Dado un grafo ponderado y dos vértices s y t se quiere hallar la distancia entre s y t, y el camino con dicha longitud. El algoritmo de Dijkstra obtiene todos los caminos de longitud mínima desde un vértice dado s al resto de vértices del grafo. El algoritmo de Floyd resuelve el problema para un par cualquiera de vértices de G.

1. Análisis de los tiempos de ejecución en forma teórica:

**Algoritmo de Dijkstra**

En cada iteración se añade un vértice a T, luego el número de iteraciones es n. En cada una se elige una etiqueta mínima, la primera vez entre n-1, la segunda entre n-2, ..., luego la complejidad total de estas elecciones es O(n^2). Por otra parte, cada arista da lugar a una actualización de etiqueta, que se puede hacer en tiempo constante O(1). Por tanto, la complejidad total del algoritmo es O(n^2).

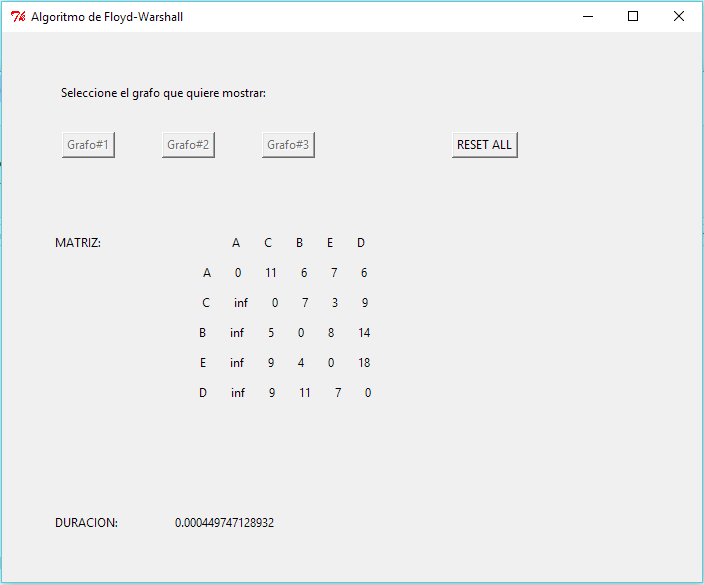
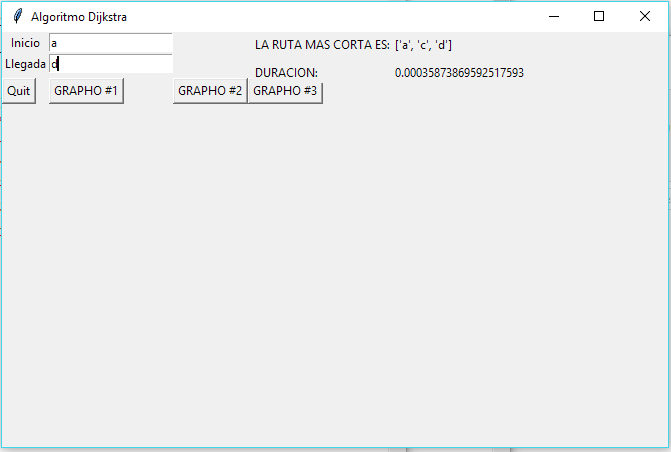
**Algoritmo de Floyd**

Se deben construir n matrices de tamaño nxn y cada elemento se halla en tiempo constante. Por tanto, la complejidad del algoritmo es O(n^3).

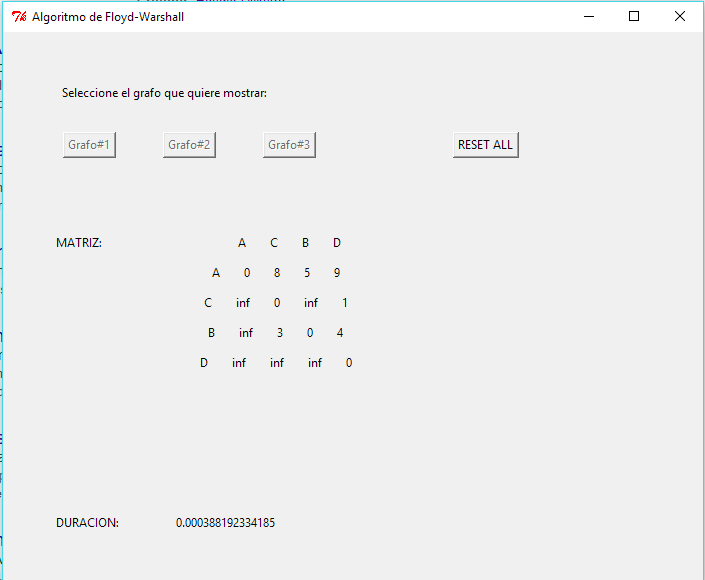
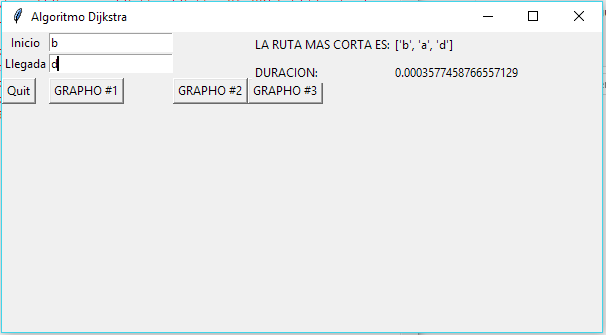
El algoritmo de Floyd es mucho más eficiente desde el punto de vista de almacenamiento dado que puede ser implementado una vez actualizado la distancia de la matriz con cada elección en k; no hay ninguna necesidad de almacenar matrices diferentes. En muchas aplicaciones específicas, es más rápido que cualquier versión de algoritmo de Dijkstra.

1. Análisis de los tiempos de ejecución con distintas entradas:

**Caso 1:**

* Floyd Grafo 1
* Tiempo: 0.000449747128932 s
* Dijkstra Grafo 1
* Tiempo: 0.00035873869592517593 s

**Caso 2:**

* Floyd Grafo 2
* Tiempo: 0.000388192334185 s
* Dijkstra Grafo 2
* Tiempo: 0.00035873869592517593 s

1. Descripción técnica:
   1. Dijkstra:

* **Algoritmo y estructura:**
* **Interfaz:**
* **Descripción:**
  1. Floyd:
* **Algoritmo y estructura:**
* **Interfaz:**
* **Descripción:**

1. Torres de Hannoi:

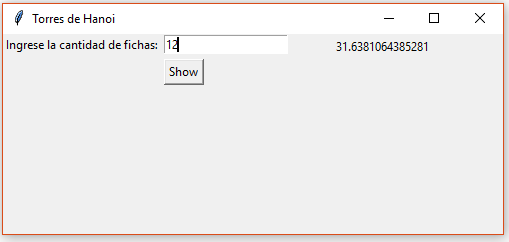
El algoritmo de las torres de Hanoi es un problema que se puede resolver mediante la recursividad, este algoritmo trata sobre discos de diferentes radios apilados del más pequeño al más grande en un base llamada torre, luego tendremos 3 torres, el objetivo de este problema es pasar todos los discos de la torre en la que tenemos a otra torre que es la objetivo, claro con diferentes restricciones. Las cuales son:

* No se puede pasar más de un plato por iteración
* No se puede poner un plato de mayor diámetro sobre uno de menor diámetro.

Cómo ya dijimos este algoritmo se resuelve forma recursiva, esto permite dividir el problema en problemas más pequeños.

1. Análisis de los tiempos de ejecución de forma teórica.
   1. En este algoritmo se pide calcular el tiempo con N muy grandes (N es el número de discos que están en las torres), Se llega a la conclusión que el caso polinomial es 2n - 1 .
   2. Sin duda el peor caso que se puede probar es donde el N es muy grande dado que la función es exponencial así que el crecimiento es realmente rápido
   3. La notación O(an)
   4. Cómo es exponencial el programa es bastante lento cuando se trata más de 20 ítems
2. Análisis de tiempos de ejecución con distintas entradas

Teniendo 12 entradas:



Teniendo 10 entradas



Cómo podemos observar la diferencia entre el resultado de 12 al resultado de 10 es bastante amplia esto es debido a que la fórmula es una exponencial, se categoriza como no polinomial.

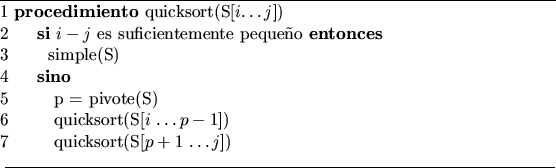
1. Quicksort y Heapsort:

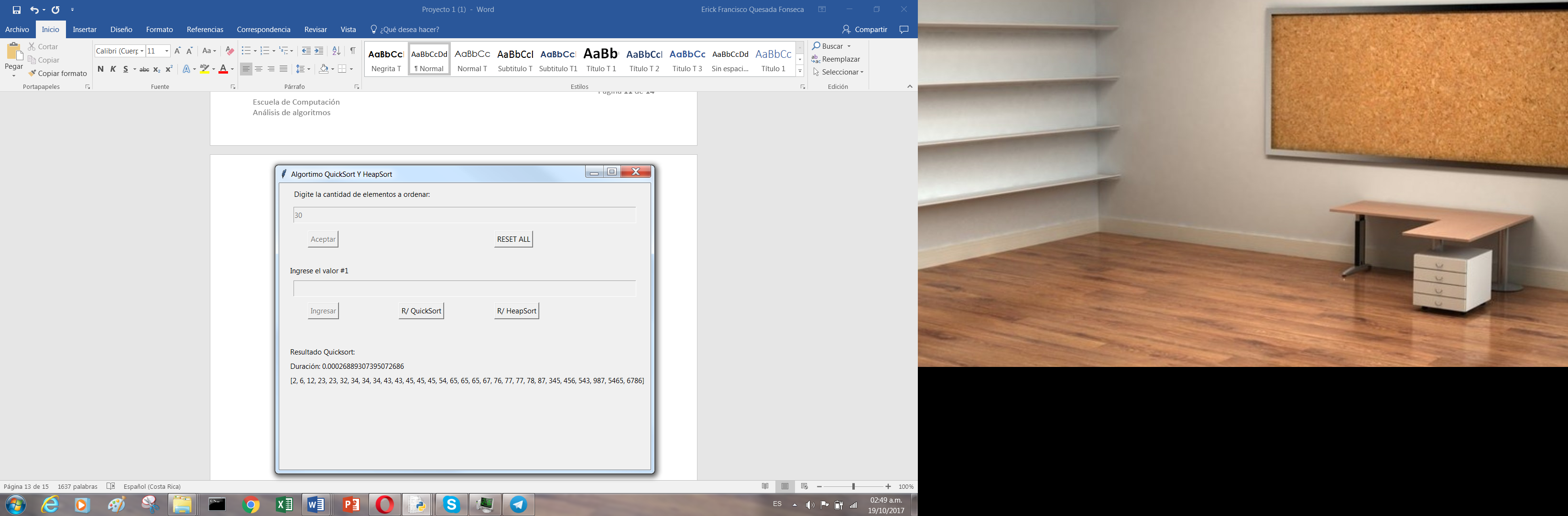
* **Quicksort**

Es el algoritmo de ordenación más rápido conocido, su tiempo de ejecución promedio es **O (n log (n))**, siendo en el peor de los casos **O(n2)**, caso altamente improbable. El hecho de que sea más rápido que otros algoritmos de ordenación viene dado por que QuickSort realiza menos operaciones ya que el método utilizado es el de partición.

1. Se elige un elemento v de la lista L de elementos al que se le llama pivote.
2. Se particiona la lista L en tres listas:
   1. L1 - que contiene todos los elementos de L menos v que sean menores o iguales que v
   2. L2 - que contiene a v
   3. L3 - que contiene todos los elementos de L menos v que sean mayores o iguales que v
3. Se aplica la recursión sobre L1 y L3
4. Se unen todas las soluciones que darán forma final a la lista L finalmente ordenada. Como L1 y L3 están ya ordenados, lo único que tenemos que hacer es concatenar L1, L2 y L3

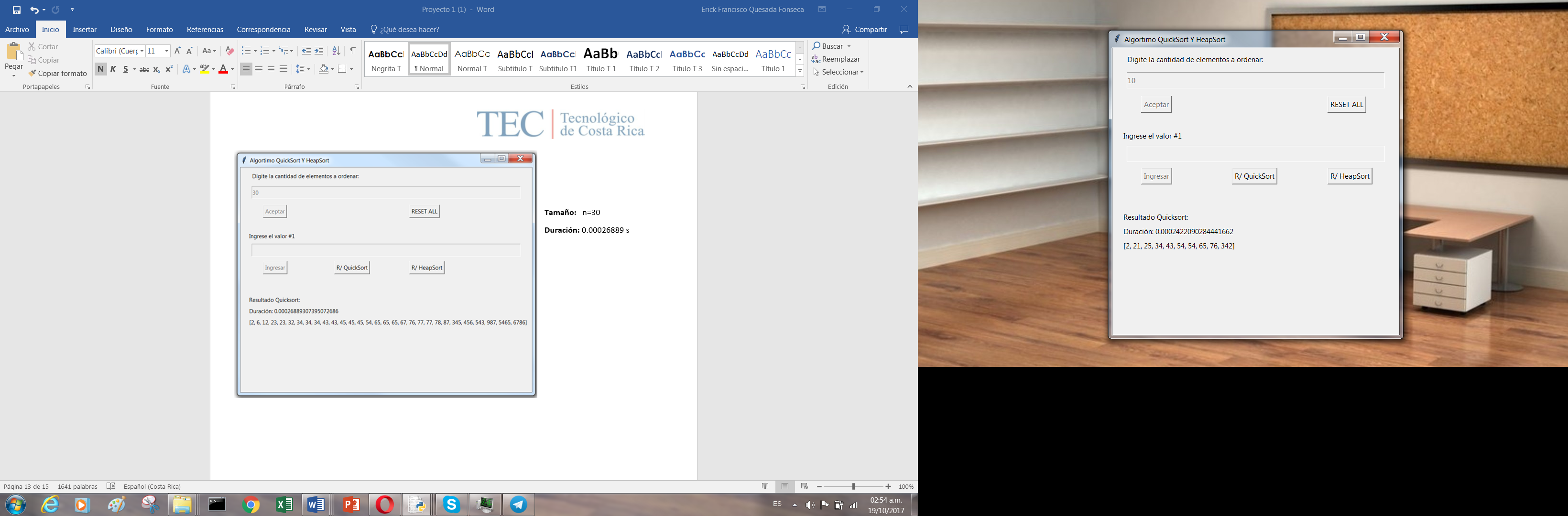
Aunque este algoritmo parece sencillo, hay que implementar los pasos 1 y 3 de forma que se favorezca la velocidad de ejecución del algoritmo.





**Tamaño:**  n=30

**Duración:** 0.00026889 s



**Tamaño:** n=10

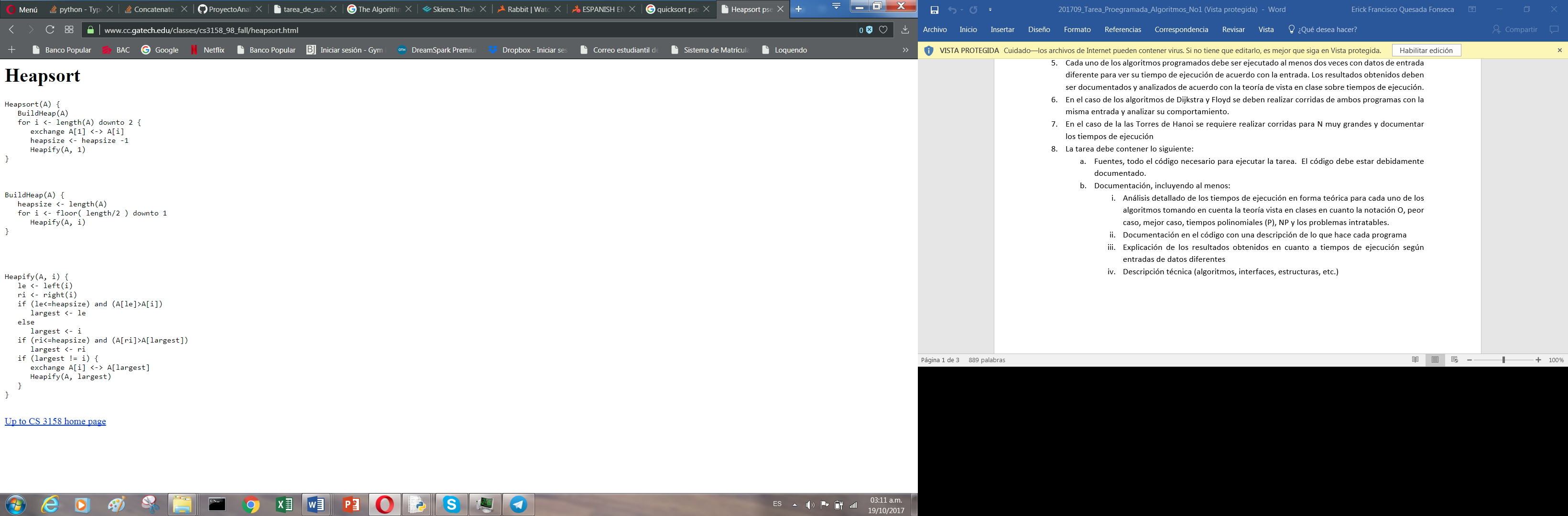
**Duración:** 0.000242209 s

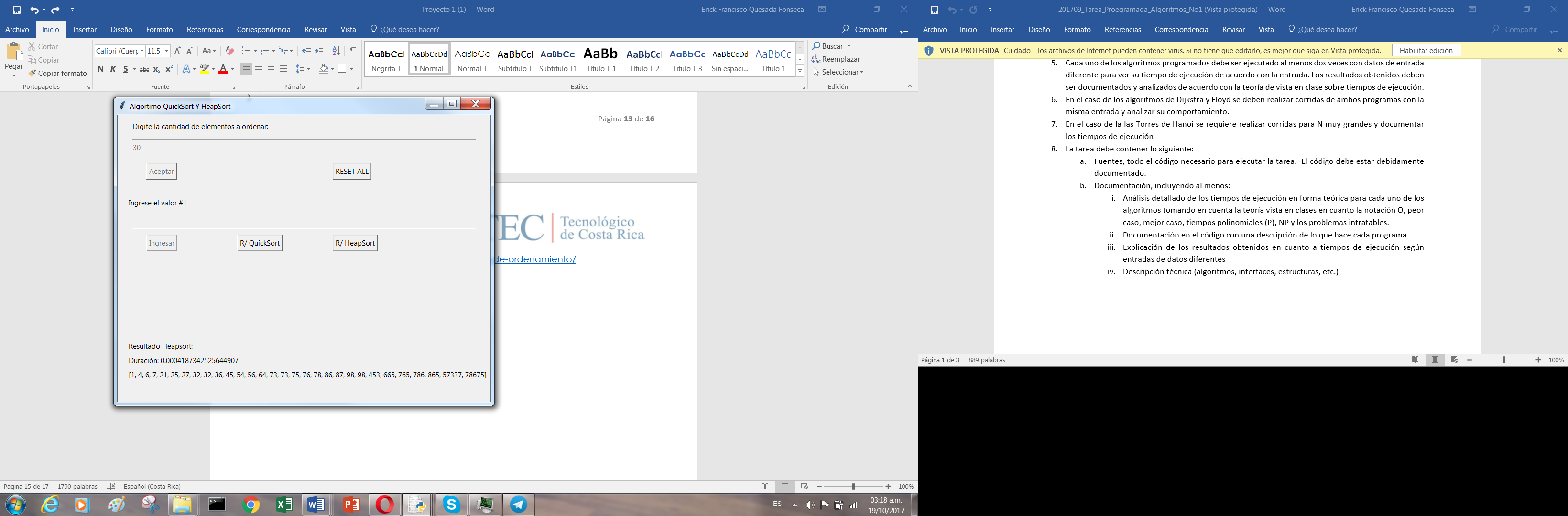
* Heapsort

Este método garantiza que el tiempo de ejecución siempre es de:

**O (n log n)**

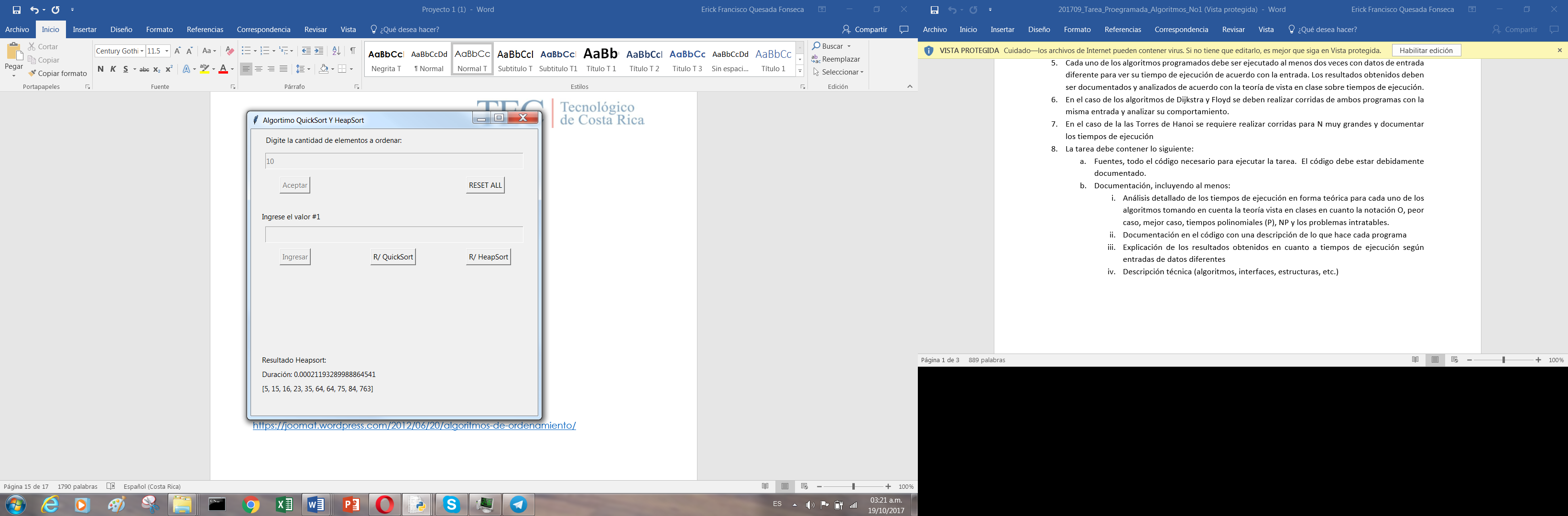
El significado de heap en ciencia computacional es el de una cola de prioridades (priority queue). Tiene las siguientes características:

1. Un heap es un arreglo de n posiciones ocupado por los elementos de la cola. (Nota: se utiliza un arreglo que inicia en la posición 1 y no en cero, de tal manera que al implementarla en C se tienen n+1 posiciones en el arreglo.)
2. Se mapea un árbol binario de tal manera en el arreglo que el nodo en la posición i es el padre de los nodos en las posiciones (2\*i) y (2\*i+1).
3. El valor en un nodo es mayor o igual a los valores de sus hijos. Por consiguiente, el nodo padre tiene el mayor valor de todo su subárbol.



**Tamaño:** n = 30

**Duración:** 0.0004187 s



**Tamaño:** n = 10

**Duración:** 0.00021193 s

1. Multiplicación de matrices en cadena:

La multiplicación de matrices en cadena nos pide solucionar el problema de cuál es la forma de realizar la multiplicación con la mínima cantidad de multiplicaciones totales posibles. Este procedimiento tiene la ventaja de ser asociativa así que sin importar el orden en el que se hagan las multiplicaciones los resultados serán los mismos. La complejidad nace en las distintas formas que se pueden realizar las multiplicaciones.

Por ejemplo tenemos 3 matrices A, B y C. Si queremos multiplicar se puede hacer de las siguientes maneras:

(AB)C

A(BC)

Ahora si tenemos 4 matrices A, B, C y D se pueden multiplicar de las siguientes maneras:

(AB)(CD)

A(B(CD))

A((BC)D)

((AB)C)D

(A(BC))D

Entonces se puede observar que solo con aumentar una matriz se eleva al doble de formas posibles de multiplicar.

La primera solución que se puede venir a la mente de una persona es probar todas las posibles combinaciones y luego de hacer esto comparar para saber de qué forma se obtiene la mínima cantidad de multiplicaciones totales. Esta solución podría hacer mucho sentido ya que sin importar que tan grande sea el número total de matrices se puede dividir y manejarse de manera recursiva. Ahora el problema con este tipo de algoritmos es el mismo problema que surge cuando se trata de solucionar la secuencia de Fibonacci de esta forma, eventualmente se van a estar resolviendo los mismos casos más de una vez y su tiempo de ejecución crecería de forma exponencial.

Para evitar este problema la investigación sobre el algoritmo llevo a la solución planteada por la página web GeeksforGeeks basados en teoría de Rashad Bin Muhammad de la universidad de Kent State. Se plantea tener una matriz de resultados temporales y en ella ir guardando los resultados de lo más básico a lo más general y así evitar el problema de tener que repetir llamadas que ya se han hecho como pasa con la solución recursiva.

1. Detalles de Implementación

Debido a que la representación de las matrices a multiplicar no se representan de la forma como se entienden que están construidas las matrices se debe explicar cómo son representadas en este algoritmo.

Para multiplicar 2 matrices se debe tener compatibilidad entre el largo de una y el ancho de la otra, estos tienen que ser iguales, entonces una matriz de tamaño aXb se puede multiplicar por una bXc pero aXb no se podría multiplicar por cXd.

Tomando esto en cuenta se representan a las matrices por medio de un arreglo de números donde una matriz es representada en el arreglo p como P[n-1] x P[n] donde n es el largo del arreglo.

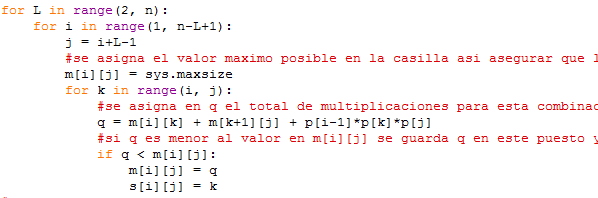
Por ejemplo si tenemos el arreglo {12, 10, 22, 34} tiene representadas 3 matrices, si n es el largo del arreglo en este caso seria 4, así que la última matriz de tamaño P[n-1] x P[n] seria de tamaño P[3] x P[4], o sea de tamaño 22 X 34, tomando eso en cuenta las 3 matrices representadas por el arreglo son de tamaño A) 12 X 10 B) 10 X 22 C) 22 X 34. Esta forma será utilizada en el algoritmo para representar las matrices.

1. Análisis de los tiempos de ejecución en forma teórica:

El algoritmo cuenta con varios ciclos y no es recursivo



El primer ciclo va de 1 hasta n así que se ejecuta N veces

  
El segundo ciclo tiene dos ciclos anidados dentro de él, el ciclo externo se ejecuta (n-2 )veces, luego el ciclo que sigue en un principio se ejecuta (n-1) veces ya que va desde un i=1 hasta un (n-L+1) que en un principio es igual a n-1 ya que L tiene un valor inicial de dos. El último ciclo va desde k=i hasta j, en un principio k=1 ya que i=1 y va hasta un J que tiene un valor inicial de (i+L-1) que tiene un valor inicial de 2 ya que i=1 y L=2. Entonces al tener estos 3 ciclos anidados el algoritmo tiene un comportamiento de θ(N3).

En cuanto a casos intratables estaríamos hablando de entradas lo suficientemente grandes para que el total de multiplicaciones sea más grande del número máximo soportado por, en este caso, Python. También cuando se tengan más entradas en el arreglo de matrices del máximo de entradas permitidas, en este caso el programa fallaría

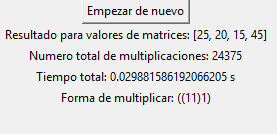
1. Análisis de los tiempos de ejecución con distintas entradas:

Caso 1:

|  |
| --- |
| **Tamaño de Matrices** |
| 25 x 20 |
| 20 X 15 |
| 15 X 45 |

Esto representado como arreglo sería {25, 20, 15, 45}

Resultado:



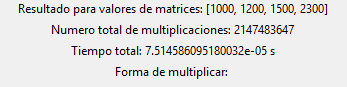
Traducción de forma de multiplicar: (AB)C

Caso 2:

|  |
| --- |
| **Tamaño de Matrices** |
| 1000 x 1200 |
| 1200 X 1500 |
| 1500 X 2300 |

Esto representado como arreglo sería {1000, 1200, 1500, 2300}

Resultado:

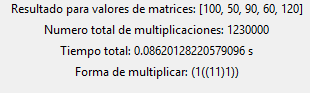
  
En este caso el número de multiplicaciones es mayor al número máximo soportado por Python que es: 2147483647, por lo tanto no representa el numero correcto si no el número máximo que puede representar

Caso 3:

|  |
| --- |
| **Tamaño de Matrices** |
| 100 X 50 |
| 50 X 90 |
| 90 X 60 |
| 60 X 120 |

Esto representado como arreglo sería {100, 50, 90, 60, 120}

Resultado:



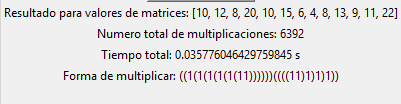
Traducción de forma de multiplicar: A((BC)D)

Caso 4:

|  |
| --- |
| **Tamaño de Matrices** |
| 10 X 12 |
| 12 X 8 |
| 8 X 20 |
| 20 X 10 |
| 10 X 15 |
| 15 X 6 |
| 6 X 4 |
| 4 X 8 |
| 8 X 13 |
| 13 X 9 |
| 9 X 11 |
| 11 X 22 |

Esto representado como arreglo sería {10, 12, 8, 20, 10, 15, 6, 4, 8, 13, 9, 11, 22}

Resultado:



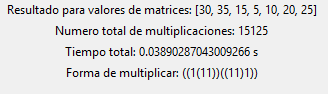
Traducción de forma de multiplicar: ((A(B(C(D(E(FG)))))))((((HI)J)K)L))

Caso 5:

|  |
| --- |
| **Tamaño de Matrices** |
| 30 X 35 |
| 35 X 15 |
| 15 X 5 |
| 5 X 10 |
| 10 X 20 |
| 20 X 25 |

Esto representado como arreglo sería {30, 35, 15, 5, 10, 20, 25}

Resultado:



Traducción de forma de multiplicar: (A(BC))((DE)F)

1. Descripción Técnica

* **Algoritmo y estructura:**

Como fue mencionado anteriormente la implementación del algoritmo viene de la página web GeeksforGeeks.com, donde se busca evitar los problemas que muchas veces se dan en programación dinámica haciendo el recorrido por la búsqueda del resultado desde abajo e ir subiendo, así evitando repetir procedimientos innecesariamente

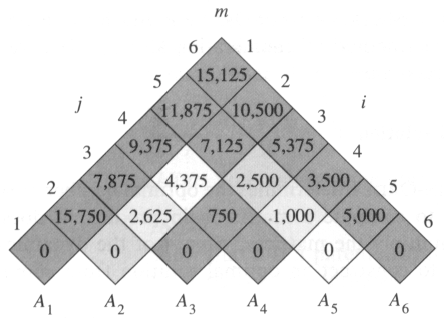
También mencionado anterior las matrices son representadas como un arreglo donde una entrada es una dimensión y la siguiente es la segunda dimensión de la matriz y así sucesivamente.

Se tiene una matriz temporal cuyas dimensiones son el largo del arreglo, la fila y columna 0 no se utilizan para facilitar el algoritmo, para propósitos de mostrar los paréntesis que muestran donde se aplica la multiplicación se tiene otro arreglo del mismo tamaño también.

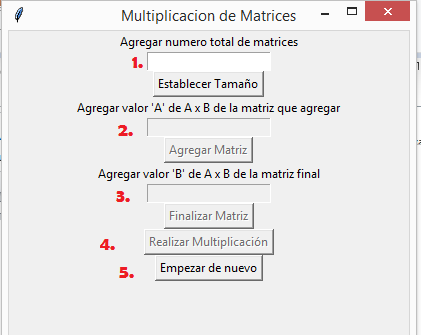
Básicamente se recorre las distintas combinaciones de multiplicaciones para cada cadena aumentando la cantidad de matrices que multiplica juntas en cada ciclo. En la matriz en cada cadena se guarda en la posición [i][j] el número mínimo de multiplicaciones para esa cadena, en ciclos posteriores estos valores son utilizados para sumar y dar el valor de multiplicaciones en otras cadenas, con esto evitar tener que calcularlos otra vez, con respecto a mostrar donde se tiene que mostrar paréntesis existe una variable k, que es la encargada del ciclo para verificar si el número de multiplicaciones es menor que el que se tiene ya registrado, entonces si esto es el caso se guarda la variable k en la posición [i][j] como lugar donde se hace va a poner un paréntesis.

Finalmente al terminar el ciclo la última respuesta de la cadena del tamaño del arreglo queda guardada en la posición [i][n-1] que si se le da vuelta a la matriz para que quede en forma de triángulo se vería como la cima de este triángulo, este valor es el total de multiplicaciones más baja para esta cadena de matrices.

Como ejemplo en el caso 5 vimos las matrices: {30, 35, 15, 5, 10, 20, 25}, en la siguiente imagen se ilustra el triángulo mencionado en el párrafo anterior y como el valor final es la cima de este:



* **Interfaz:**

****

* **Descripción:**

1. Lo primero que se establece es la cantidad total de matrices que se van a multiplicar, aquí el usuario digitara un número y al darle establecer tamaño podrá empezar a agregar matrices

2. El usuario agrega el tamaño de cada matriz recordando que las dimensiones van seguidas como se mencionó previamente, se podrán agregar matrices hasta llegar a tener el número que se digito anteriormente, cada registro es reflejado en la pantalla

3. Al insertar la última matriz se habilitara el tercer campo de texto donde se digita la última dimensión de la última matriz, una vez hecho esto se da click en finalizar matriz y se puede pasar al siguiente paso

4. Se realiza la multiplicación de matrices al apretar el botón, se ocupa tener todas las matrices que se establecieron cuando se agregó el total, si no esta opción no será habilitada, el programa desplegara la cantidad de multiplicaciones, el tiempo que duró y la forma como se debe multiplicar las matrices para obtener esto. La forma de leer lo escrito es de izquierda a derecha asignando un valor A al primer 1 que se vea, B al segundo y así sucesivamente hasta que todas las matrices tengan un valor

# Preguntas:

1) ¿Funciona el algoritmo de Floyd en un grafo que tenga algunas aristas cuyas longitudes sean negativas pero que no contengan ningún ciclo negativo? Cuál es la relación con los problemas clase P, NP o NP completos. Justifique.

Sí funciona, sin embargo, si permitimos que el grafo tenga aristas con valores negativos, el algoritmo de Floyd pierde parte de su significado al buscar le camino más corto. No se conoce un algoritmo eficiente que halle los caminos simples más cortos en este tipo de grafos, este problema se clasifica como NP completo, a diferencia del algoritmo solamente con aristas positivas, este se resuelve en un tiempo polinomial.

2) Bajo qué circunstancias el algoritmo de Floyd es mejor al de Dijkstra. Justifique.

El algoritmo de Floyd tiene una complejidad de O(V^3), y con esto encuentra las mejores rutas para todos los pares, mientras que Dijkstra tiene una complejidad de O(E + Vlog(V)). Al correr el algoritmo de Dijkstra sobre todos los vértices, para poder encontrar todas las mejores rutas, la complejidad se convierte en O(EV + V^2log(V)). E representa la cantidad de aristas, por lo tanto, si E es del orden O(V^2) Dijkstra va a ser tan rápido como Floyd, sin embargo, si E es del orden O(V), el tiempo de ejecución del algoritmo de Dijkstra es menor.

Por lo tanto, para un grafo altamente denso Floyd va a tener un mejor rendimiento, además que se puede utilizar con aristas de valor negativo y Dijkstra no.

3) ¿Cuáles serían los tiempos de respuesta estimados para el algoritmo de las Torres de Hanoi con N = 50 discos y corriendo en una máquina que procesa diez mil millones de instrucciones por segundo? Justifique.

El algoritmo de Torres de Hanoi tiene una complejidad de O(2^N), siendo N la cantidad de discos.

Por lo tanto, tomando N = 50, la complejidad será de 1125899906842624.

La fórmula para obtener el tiempo de ejecución es la siguiente:

complejidad /cantidad de instrucciones por segundo = tiempo de ejecución en segundos

1125899906842624 / 10000000000 = 112589.990 s

112589.990 s = 1876.50 m = 31.275 h

El tiempo de respuesta del algoritmo de Hanoi es de 31.275 h, cuando se usan 50 discos.

Referencias:

[www.lab.dit.upm.es/~lprg/material/apuntes/o/index.html](http://www.lab.dit.upm.es/~lprg/material/apuntes/o/index.html)

<https://en.wikipedia.org/wiki/Knapsack_problem>

<https://es.wikipedia.org/wiki/Problema_de_la_mochila>

<https://es.wikibooks.org/wiki/Algoritmia/Programaci%C3%B3n_din%C3%A1mica>

<https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=8&ved=0ahUKEwjct7H9n_zWAhVOySYKHagOAhIQFghVMAc&url=https%3A%2F%2Fwww.dc.uba.ar%2Fmaterias%2Faed3%2F2013%2F1c%2Fteorica%2Fdinamica.pdf&usg=AOvVaw22z65C9QjnwVeE4ro9Uy6f>

<https://joomat.wordpress.com/2012/06/20/algoritmos-de-ordenamiento/>

<https://www.genbetadev.com/algoritmos/implementando-el-algoritmo-quicksort>

<https://es.wikibooks.org/wiki/Programación_dinámica/Problema_de_las_monedas_con_programación_dinámica>